Proceedings of the 53rd Hawaii International Conference on System Sciences | 2020

Evolving Neural Networks to Solve a Two-Stage Hybrid Flow Shop Scheduling Problem with Family Setup Times

Sebastian Lang
Fraunhofer Institute for Factory Operation and Automation IFF
sebastian.lang@iff.fraunhofer.de

Tobias Reggelin
Institute of Logistics and Material Handling Systems
Otto von Guericke University Magdeburg
tobias.reggelin@ovgu.de

Fabian Behrendt
Fraunhofer Institute for Factory Operation and Automation IFF
fabian.behrendt@iff.fraunhofer.de

Abdulrahman Nahhas
VLBA Lab
Otto von Guericke University Magdeburg
abdulrahman.nahhas@ovgu.de

Abstract

We present a novel strategy to solve a two-stage hybrid flow shop scheduling problem with family setup times. The problem is derived from an industrial case. Our strategy involves the application of NeuroEvolution of Augmenting Topologies - a genetic algorithm, which generates arbitrary neural networks being able to estimate job sequences. The algorithm is coupled with a discrete-event simulation model, which evaluates different network configurations and provides training signals. We compare the performance and computational efficiency of the proposed concept with other solution approaches. Our investigations indicate that NeuroEvolution of Augmenting Topologies can possibly compete with state-of-the-art approaches in terms of solution quality and outperform them in terms of computational efficiency.

1. Introduction

The two-stage hybrid flow shop scheduling problem (THFS) is a well-investigated combinatorial optimization problem in production and logistics. It describes a system with two production stages, where minimum one stage has more than a single machine and where each job can be processed on each machine [1]. The objective is to determine a job sequence and an allocation of jobs to machines that minimize a given objective function. Even in its simplest form (a system consisting of one stage with a single machine and another stage with two machines), the hybrid flow shop scheduling problem is proven to be NP-hard [2].

There are several variants and specification of the THFS. Ruiz and Vázquez-Rodríguez (2010) give an overview of the problem and common solution strategies for the different specifications [3]. The problem addressed in this paper is a THFS with family setup times, which meant that a machine might require additional time for preparation before processing a job. The problem is based on an industrial case study of a printed circuit board (PCB) assembly. Aurich et al. (2016) already solved this problem with Tabu Search (TA), Simulated Annealing (SA) and a self-developed integrated simulation-based optimization (ISBO) heuristic [4]. They compared their results with the performance of a setup minimizing family production strategy (FP), applied by the analyzed company, and with the priority dispatching rules Shortest Processing Time (SPT) and Earliest Due Date (EDD). As expected, the metaheuristics and the self-developed heuristic were able to achieve better results than the priority dispatching rules.

However, both metaheuristics and the self-developed heuristic (ISBO) have their drawbacks. On a CPU with 4x2.6GHz, the computational time of the metaheuristics for calculating a sufficient schedule for about 160 jobs is between 30 and 45 minutes. Due to the high dynamics of the company's production environment, feasible scheduling decisions are required in much shorter time. The ISBO finds approximately the same solutions as the metaheuristics in 8 to 18 seconds and thus fulfills the requirements of the company. However, as the authors tailored the ISBO on the specifications of the company's production environment, its adaption to other problems is a tough task and requires a certain knowledge in developing optimization heuristics.

In this paper, we present a novel strategy for tackling scheduling problems in production and logistics, which overcomes the mentioned drawbacks. We combine NeuroEvolution of Augmenting Topologies together with discrete-event simulation (DES) to generate neural networks, which estimate

URI: https://hdl.handle.net/10125/63899
978-0-9981331-3-3

HICSS

production schedules based on job attributes and states of the production system. To the best of our knowledge, no related research describes a similar approach to solve scheduling problems in production and logistics.

The further paper is organized in five sections. Section 2 summarizes the main ideas of NeuroEvolution of Augmenting Topologies. Section 3 provides an overview of the adoption of neuro-evolution and comparable approaches for solving scheduling problems. Section 4 describes the problem and our solution strategy. In Section 5, we present our experiments and results. We further compare our results with the previous study [4]. Section 6 is dedicated to the conclusions.

2. NeuroEvolution of Augmenting Topologies

NeuroEvolution of Augmenting Topologies (NEAT) is a genetic algorithm (GA) that evolves the topology and hyper-parameters of neural networks to find the best configuration for a given machine-learning task.

A GA is a metaheuristic, which searches for solutions to a given optimization problem by imitating the process of natural evolution. In particular, a GA initially creates a random set of start solutions (population) and randomly modifies (mutates) single solution vectors (genomes) as well as randomly recombines several solution vectors to new ones. This process is repeated over several iterations (generations), until a predefined termination criteria is met (e.g. identification of a solution that fulfills certain quality criteria) [5].

Stanley and Miikkulainen presented NEAT for the first time in 2002 [6]. They provide a detailed description of NEAT in [7]. NEAT is comparable to reinforcement learning as it does not require labeled training data for learning. Instead, the algorithm improves the parameters of a neural network based on feedback signals of a fitness function. The algorithm is considered to be the first neuro-evolution strategy, which can efficiently evolve the topology of neural networks, due to the utilization of three techniques:

1. The algorithm tracks for every genome its origin. The authors observe that genomes with different topologies can crossover in a meaningful way, if they originate from the same ancestral genome.
2. NEAT divides the total population in different species depending on topological similarities. In consequence, genomes only compete with others of their own species. Thus, a neural network that has optimized its hyper-parameters over several generations can still evolve its topology without being instantly removed from the population, because of a fitness loss.
3. Comparable approaches suffer from poor computational performance, because they consider different topologies in the initial population. That leads to a high dimensionality of the genomes even before the first iteration starts. As discussed above, the consideration of different topologies in the initial population is for many neuro-evolution approaches necessary, as a significant change of the topology in later generations lead to an initial fitness loss. However, due to the division of the population in different species, NEAT does not require to generate a large initial population. Thus, initial genomes have no hidden neurons and differ only in terms of their hyper-parameters. NEAT mutates the topology of the neural networks incrementally and only those topologies survive, which can compete with other genomes of the same species. During the evolution process, NEAT does not restrict the number of hidden layers and neurons and is therefore applicable for problems of any complexity.

For our research, we use the open-source library neat-python [8].

3. Related work

It appears difficult to find many publications that describe the application of neuro-evolution approaches (including NEAT) for scheduling problems. In fact, we were only able to investigate three publications of which only a single paper addresses production scheduling [9]. The paper describes a scheduling problem with identical parallel machines, which is considered fairly less complicated than a THFS. Furthermore, the authors apply a neuro-evolution approach, which is only able to adjust the weights of a predefined neural network structure. The other two publications discuss the application of neuro-evolution for resource-allocation on chip-multiprocessors [10] and for scheduling jobs on severs [11]. Both problems are less complicated than a THFS, because they can also interpreted as scheduling problems with identical parallel machines.

As discussed in the previous section, NEAT shares many similarities with reinforcement learning (RL). Therefore, we also want to give a brief overview of publications adopting RL for scheduling problems in production and logistics. Our findings reveal that Zhang and Dietterich (1995) describe the application of RL for job-shop scheduling for the first time [12]. In contrast to our approach, the authors apply RL for the evaluation and not for the determination of job schedules. We consider the paper of Aydin and Öztemel (2000) as the first paper, which describes an RL-agent whose actions influence the scheduling of jobs [13]. However, job schedules are not defined directly as in our approach. Instead, the agent chooses from a set of priority dispatching rules the best believed alternative for a specific system state. Comparable approaches to [13] are described in [14, 15]. Paternina-Arboleda and Das (2005) propose the adoption of RL to determine a dynamic control policy for a stochastic lot scheduling problem on a single machine. To specific system states, the agents decide to which setup type the machine shall be configured [16]. Qu et al. (2016) pursue a similar approach for a multi-stage flow shop problem. They also consider information regarding the maintenance of machines and regarding the condition of workers. Thus to decide, whether a machine shall change to a specific setup type [17]. Nonetheless, both papers do not consider the application of RL for the direct generation of job schedules. The first paper we found that describes RL agents being able to directly allocate and sequence jobs is from Stricker et al. (2018). The different agent types are responsible for different production control decisions, such as selecting the next job to be processed or assigning the selected job to a machine [18]. The authors compare their solution approach with a FIFO dispatching strategy. In terms of system utilization, the RL-agent outperforms FIFO by around 10%. Furthermore, Waschneck et al. (2018) propose a combination of supervised learning and deep RL for job-shop scheduling in a semiconductor production [19]. They compare the performance of their approach with an event handler, which operates based on expert knowledge. However, it is not clear how the agents affect the scheduling of jobs exactly, because the paper contains only little information about the action spaces of the agents.

As conclusion of the related work and with respect to the mentioned drawbacks of related optimization approaches in the introduction, we want to summarize the main contributions of this paper:

1. The paper presents for the first time the application of NEAT in conjunction with DES for a production scheduling problem. As NEAT is a metaheuristic, the method can be applied for any combinatorial optimization problem, without requiring in-depth knowledge about the problem itself. NEAT approaches an optimization problem indirectly by optimizing the representation of a neural network, which solves the problem in appropriate way. Therefore, we expect a very low computational time to calculate solutions for similar problem instances.
2. We introduce a concept to encode allocation and sequencing problems as machine-learning tasks. The concept allows the creation of neural networks, which are able to estimate production schedules based on a given set of input data.
3. We evaluate the performance of our approach on a real industrial use case in terms of solution quality and computational efficiency. We further compare our results with two metaheuristic approaches (SA and TS), a self-developed heuristic (ISBO) and three priority dispatching rules (FP, SPT, EDD). By this means, we will show that our approach provides a trade-off between computational costs and solution quality.

4. Proposed solution strategy

In this section, we present our solution strategy for solving the THFS with family setup times. First, we will formulate the problem and the objective function. Second, we discuss two alternatives to formulate the presented THFS as a machine-learning task and derive the basic neural network design. Third, we describe the integration of NEAT and the DES model.

4.1. Formulation of the problem

In the following, we will summarize the characteristics of the analyzed THFS. Our summary is based on the comparative study [4].

The considered THFS has two production stages. The first stage consists of four identical parallel surface mount device (SMD) placement machines. The second stage has five identical parallel automated optical inspection (AOI) machines. The capacity of the queues in front of the machines is negligible (i.e. infinite). Each job has to be processed on a single machine of each production stage. A job j is defined by its:

- due date d_j (time unit: minutes)
- job family s_j
- process time on the first stage $t_{SMD,j}$ (time unit: minutes)
- process time on the second stage $t_{AOI,j}$ (time unit: minutes)

The processing times $t_{SMD,j}$ and $t_{AOI,j}$ are specific for each job. The machines of the first production stage underlie major (65 minutes) and minor setup times (20 minutes). An SMD placement machine requires a major setup, if the family of the job to be processed s_j is different from the family of the job previously processed. A minor setup is required, if the families of two consecutive jobs are the same. AOI machines of the second stage underlie a general setup time of 25 minutes. The input data consists of four different datasets, which differ in terms of the number of jobs and the parameters of the jobs. Table 1 provides an overview about the characteristics of each dataset. The complete datasets are available on our website[1].

We further define the problem as a permutation flow shop. For a hybrid flow shop problem, this means that each job is released on the first stage according to an initial job sequence. On successive stages, however, jobs will be allocated to the earliest time that a machine of that stage becomes available [1]. The consideration of the problem as permutation flow shop corresponds to the production process of the company. This is justified by the fact that control decisions for SMD placement machines have the most impact on the performance of PCB assembly lines [20]. Consequently, our solution strategy will focus on the first production stage.

Finally, we consider the THFS as a deterministic problem. Therefore, we do not take into account stochastic influences, such as arbitrary machine breakdowns.

After describing the system and processes, we now want to introduce the underlying optimization problem. The objective of the optimization is to find a schedule, which reduces the total tardiness T and the makespan C_{max}.

Table 1. Input datasets (based on [4])

	D 1	D 2	D 3	D 4
$\#jobs$	164	170	175	143
$\#job\ families$	41	37	36	35
$due\ date$	1305–27405	1305–27405	1305–27405	1305–27405
t_{SMD}/job	4–3142	2–3736	4–3293	4-3209
$\sum t_{SMD}$	54685	62345	61274	56250
t_{AOI}/job	4–4351	3–5590	5–3528	3–4300
$\sum t_{AOI}$	72528	88702	74738	79294

The total tardiness T is the sum of tardiness over the number of all jobs n [21], as described by formula (1).

$$T = \sum_{j=1}^{n} T_j \tag{1}$$

The tardiness of a single job T_j is the lateness of a job, if the job is completed after its due date. The lateness of a job is the difference between its completion time C_j and its due date d_j [21], as described by formula (2).

$$T_j = \begin{cases} c_j - d_j & if\ (c_j - d_j) > 0 \\ 0 & otherwise \end{cases} \tag{2}$$

The makespan C_{max} is the maximum completion time over the number of all jobs n [21], as described by formula (3).

$$C_{max} = \max(C_j\ \forall\ j \in \{1, \ldots, n\}) \tag{3}$$

In order to minimize both, the total tardiness and the makespan, we adapt the weighted sum approach to formulate the objective function.

$$minimize\!: \quad \gamma_1 * T + \gamma_2 * C_{max} + \gamma_3 * n_s \qquad \gamma_1 > \gamma_2 \geq \gamma_3 \tag{4}$$

…where n_s is the number of major setups and γ_1, γ_2 and γ_3 are the weights of T, C_{max} and n_s respectively. The weight constrained is subject to the company's preference that the minimization of the total tardiness is more important than the makespan. Furthermore, the number of major setups is not an optimality measure and therefore negligible as long as a job family is only processed on a single SMD machine at the same time (due to the limited number of setup carts). However, the comparative study [4] considers the number of major setups assuming that a reduction leads also to a reduction of the makespan. This assumption seems legitimate, as the time required to change the setup of a machine is at the expense of the available time to process jobs. Therefore, we initially consider the number of major setups in our objective function.

[1] https://www.ilm.ovgu.de/hicss_problem_instances.html

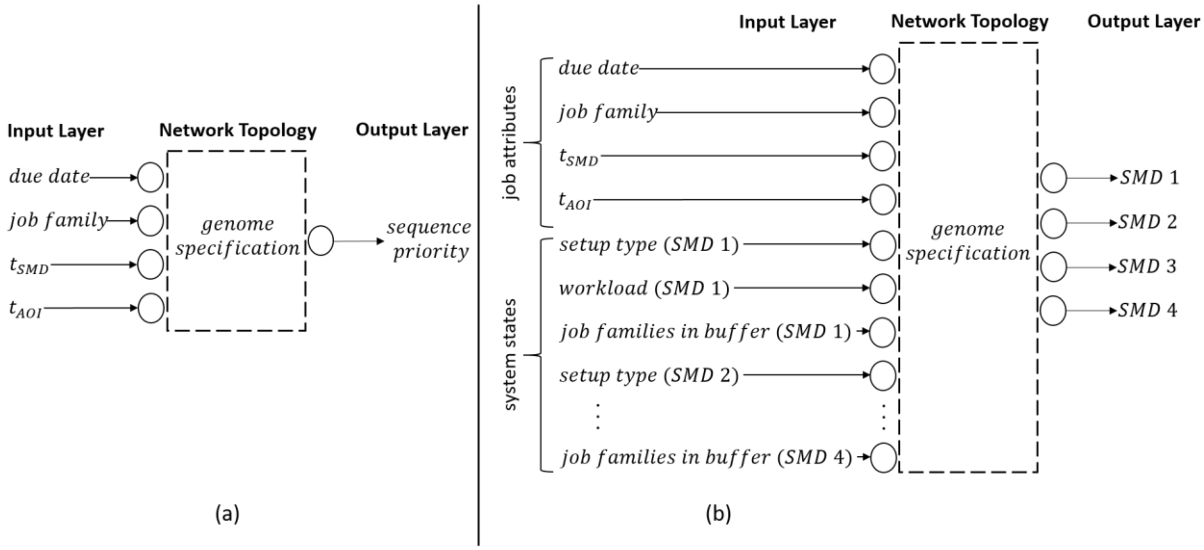

Figure 1. Encoding of the networks input and output for (a) sequencing problems (regression) and (b) allocation problems (classification)

We implemented the THFS as DES model to evaluate the genomes of NEAT. The DES model provides training signals for NEAT by computing the fitness, i.e. the objective function, for each genome. We decided for the open-source python library salabim [22] as simulation framework, because it allows the integration of NEAT and the DES model without any additional communication interfaces.

4.2. Formulation of the machine-learning task

Any hybrid flow shop problem can be reduced to a certain number of sequencing and allocation problems. Considering only the first stage of the THFS, the optimization can be narrowed down to one allocation problem (i.e. the distribution of jobs on four SMDs) and one or five sequencing problems (sorting the jobs in the source or in each SMD buffer). The number of sequencing problems depends on the existence of a permutation constraint, which do not allow the alternation of job sequences during simulation.

As figure 1 illustrates, the machine-learning task and the possibilities for the basic neural network design, i.e. the encoding of the networks input and output, are different for sequencing and allocation problems. The job attributes are primarily relevant for the sequencing of jobs. If the permutation of job sequences during simulation is allowed, the SMDs setup type could be also relevant as input parameter for the network. The sequencing of jobs can be formulated as regression problem. Depending on the attributes of jobs, the neural network iteratively outputs numbers, which are used to prioritize jobs against each other.

The allocation of jobs on the other hand corresponds to a classification problem, where each output neuron represents an allocation option. Theoretically, it is also possible to formulate the sequencing problem as a classification problem, in which the number of jobs corresponds to the number of output neurons. However, the training of a classifier is only meaningful, if the number of jobs to be scheduled is always the same. This does not apply for our case. Beside job attributes, the allocation of jobs also requires the consideration of system states (e.g. buffer utilization, SMD workload, number of different job families in the SMDs buffer, etc.), as allocation decisions directly affect the state variables of the system. Table 2 on the following page presents three strategies to solve the THFS at hand by considering only the first stage.

The rating of the training effort and size of the evaluable solution space is based on subjective assessments. However, it seems obvious that the second strategy leads to a higher training effort as the first, because with increasing number of input/output-relations, NEAT has to evaluate more network configurations. The third strategy requires the highest training effort, as it requires the execution of at least six NEAT sessions to find appropriate network configurations for the allocation problem and the five sequencing problems. In the further course, we will only present experiments to the first strategy, which is appropriate for providing a first proof of concept to our idea.

Table 2. Solution strategies to solve the THFS with neural networks

Strategy	Description	Characteristics
Solving the THFS as sequencing problem with permutation constrained	A regressor network determines an initial job sequence, before the simulation starts	• training effort: low • solution space: limited
Solving the THFS as allocation problem	A classifier network allocates jobs to SMDs during the simulation	• training effort: medium • solution space: middle
Solving the THFS as allocation problem and sequencing problem	A classifier allocates jobs to SMDs during the simulation; several regressor networks adapt the job sequences in the SMD buffers during simulation	• training effort: high • solution space: large

4.3. Integration of NEAT and the DES model

Figure 2 on the following page illustrates how NEAT and the DES model are integrated. In the following, we will shortly describe how the different components are interacting with each other.

In an initial step, a script imports the list of jobs to be scheduled. Training features are scaled and transformed in value ranges between zero and one in order to avoid a misguidance of NEAT's search caused by different data dimensions.

Afterwards, the NEAT algorithm is initialized. NEAT reads first the experiment parameters from the configuration file and creates an initial population. The algorithm initiates thereafter the iterative search for the best network configuration. NEAT creates a neural network from each genome in the population and transmit it to the DES model.

Before a simulation experiment starts, the current evaluated neural network sequentially processes the feature list and assigns a priority index to each job. Depending on the activation function of the output neuron, the priority index is a real value between 0 and 1 (sigmoid function) or -1 and 1 (hyperbolic tangents (tanh) function). The sequencing function of the simulation model sorts the jobs by descending priority index.

In the next step, the DES model initiates the simulation with the generated job sequence. As we only want to analyze the performance of a neural network on the sequencing problem with permutation constraint, the simulation model make use of two simple rules for the allocation of jobs on both stages. For an incoming job on the first stage, the model checks for each SMD, whether the family of the last allocated job is the same as the family of the job to be allocated next. The model allocates the job to the first found SMD that fulfills the condition. In general, this condition is necessary to avoid that two SMD machines have the same setup type at the same time. As discussed in section 4.1, such a state is not allowed, due to a limitation of setup carts in the real system. If no SMD fulfills the condition, the model assigns the job to the SMD with the lowest workload. Likewise, the model allocates arriving jobs on the second stage directly to the buffer with the lowest workload.

After a simulation run is finished, the model computes the fitness of the corresponding genome according to the objective function. As soon as the DES model evaluated the complete population, NEAT evolves the genomes' properties and goes into the next iteration.

5. Experiments and results

We divided the four datasets into two groups. Dataset 2 and dataset 3 were considered as training sets, i.e. the datasets for the evaluation of genomes. We decided for dataset 2 and dataset 3, because they provide the highest number of training samples. Dataset 1 and dataset 4 represent our test sets, on which we validated the performance of the winner genome.

We ran NEAT several times to fine-tune the configuration parameters. We initially used the configuration file of the XOR example from the neat-python project website [7] with two changes. First, we enabled the creation of recurrent connections to allow the generation of recurrent neural network structures beside feedforward multiple layer perceptrons. Second, beside the sigmoid function, we also considered tanh as possible activation function. With the initial configuration, NEAT rapidly converged to a specific net structure. The resulting network, however, was not yet able to generate production schedules in sufficient quality. Consequently, we increased the mutation rates, which led to better results. The parameters addressing the stagnation of species and the reproduction of genomes seem to have the highest influence on the algorithm's dynamics (i.e. the number

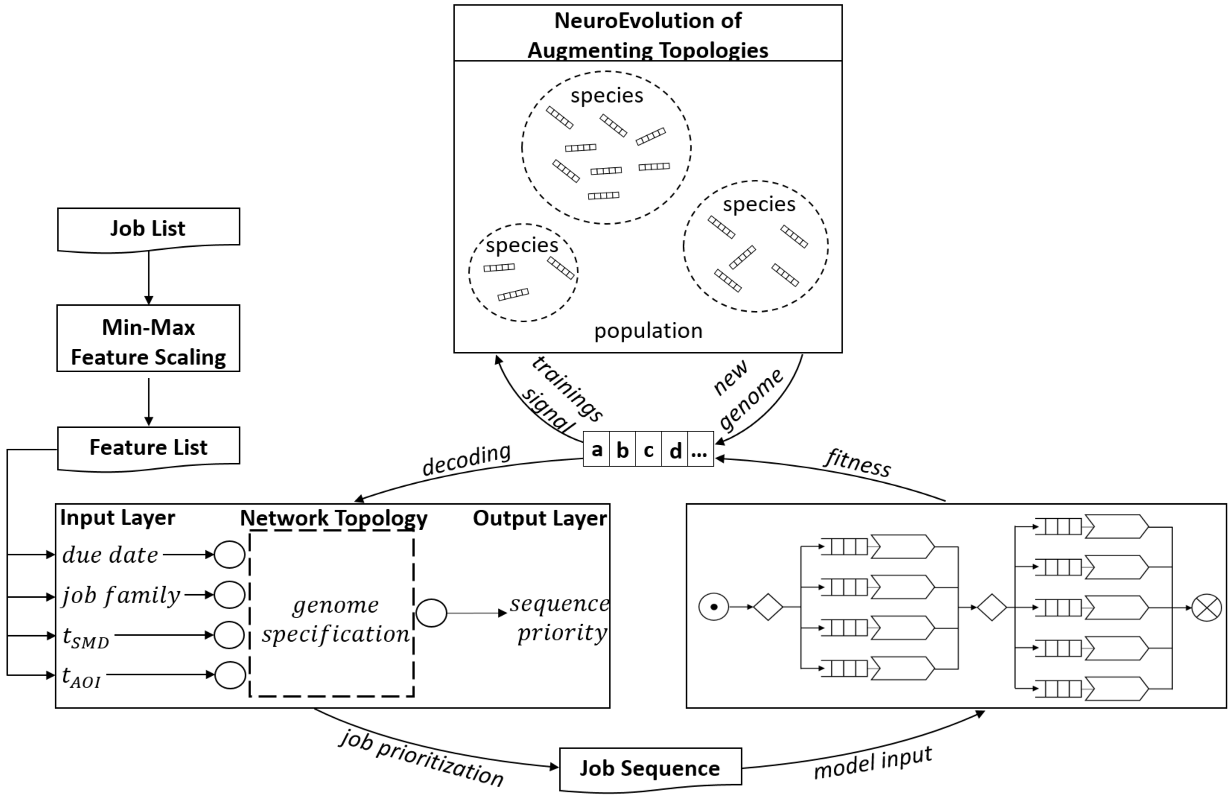

Figure 2. Conceptual model of our solution strategy

of generations after a species is eliminated caused by stagnation, the number of species, which are protected from elimination caused by stagnation, the number of the most-fit genomes in each species, which are protected from mutations). Reducing these parameters to smaller values significantly improved the convergence behavior of NEAT.

Furthermore, we tested several objective functions, taking into account the total tardiness, the makespan and the number of major setups with varying weights. Our initial weights were γ_1 = 0.7 (total tardiness) and γ_2 = γ_3 = 0.15 (makespan and number of major setups). However, after several experiments, we identified the total tardiness, without the makespan and the number of setups, as best fitness function ($\gamma_1 = 1, \gamma_2 = \gamma_3 = 0$). Table 3 on the following page presents the optimization results of NEAT for each dataset in comparison to the investigations of [4]. Table 4 provides a summary of the results.

Like ISBO, SA and TS, NEAT clearly dominates the priority dispatching rules in terms of the minimization of makespan and total tardiness. In the same way, NEAT also slightly outperforms the ISBO. However, only the metaheuristics SA and TS were able to find solutions for every dataset, which do not violate any due date. In our opinion, the results provide three remarkable insights:

1. NEAT outperforms the other approaches in terms of makespan optimization. This is surprising as the best performing version of the algorithm only considers the total tardiness as fitness function.
2. When we decided to consider only the total tardiness as objective function, we assumed that NEAT would converge to a neural network that approximates the behavior of the EDD dispatching rule. As table 4 shows, NEAT clearly outperforms EDD in both, makespan optimization and total tardiness optimization.
   First investigations indicate that minimizing the number of major setups is less effective for reducing the makespan than just minimizing the total tardiness. For instance, we set $\gamma_3 = 0.15$ and were able to reduce the number of major setups from about 130 down to 100. For the training datasets D2 and D3, we achieved results for makespan and total tardiness that are comparable to table 3. However, the same network performed dramatically worse on the test datasets. A possible explanation could be that the neural network overfits the training data, if

the objective function considers too many parameters. By this means, the intuition arises that a simpler objective function leads to a better generalization of the training data. However, this hypothesis requires a comprehensive parameter study, which will be subject of future investigations.

Table 3. Performance of NEAT for each dataset in comparison to [4]

	Total tardiness (minutes)	Makespan (minutes)	Number of setups
Dataset 1			
FP	198,783	23,513	37
SPT	86,490	23,586	126
EDD	0	21,154	104
ISBO	148	19,354	43
SA	0	21,930	45
TS	0	19,669	45
NEAT	**124**	**17,768**	**114**
Dataset 2			
FP	271,700	25,447	33
SPT	149,141	26,662	135
EDD	4,833	26,226	136
ISBO	0	21,819	53
SA	0	23,108	55
TS	0	25,142	55
NEAT	**303**	**20,916**	**149**
Dataset 3			
FP	31,372	23,626	32
SPT	149,148	25,756	131
EDD	6,000	22,603	139
ISBO	536	19,979	56
SA	0	23,059	59
TS	0	22,507	60
NEAT	**0**	**20,584**	**142**
Dataset 4			
FP	257,376	23,539	31
SPT	11,518	20,507	113
EDD	964	21,145	113
ISBO	639	18,806	42
SA	0	20,562	58
TS	0	21,610	57
NEAT	**0**	**18,771**	**113**
FP - Family Production (company strategy) | SPT - Shortest Processing Time | EDD - Earliest Due Date | ISBO - Integrated Simulation Based Optimization | SA - Simulated Annealing | TS - Tabu Search			

Table 4. Average performance of NEAT in comparison to [4]

	Avg. total tardiness (minutes)	Avg. makespan (minutes)	Avg. number of setups
FP	162,510.33	24,031.25	33.25
SPT	103,269	24,127.75	126.25
EDD	1,450.75	22,782	123
ISBO	330.75	19,989.5	48.5
SA	0	22,243	54.25
TS	0	22,439.33	54.25
NEAT	**106.75**	**19,509.75**	**129.5**

In summary, we are positively surprised about the performance of NEAT, as we expected a lower solution quality against an improvement of the computational time. As table 4 and 5 show, NEAT achieves both, competitive results and an outstanding computational efficiency. For the test datasets, the best-found neural network generates and evaluates a job sequence in 0.1 to 0.2 seconds. Therefore, the proposed concept offers great potential for the design of real-time capable decision-support systems. The computational time until NEAT converges to a preferred network structure is comparatively low as well. With an initial population size of 300 genomes, NEAT usually requires less than 15 generations to identify the best neural network configuration for the sequencing problem with permutation constraint. We also conducted several experiments, in which we evaluated 500 to 600 generations. However, figure 3 on the following page shows that with increasing number of generations, the algorithm slips out of the search range of the best solution. On the one hand, this observation indicates that solving the sequencing problem with neural networks requires only very simple network topologies, because NEAT initializes neural networks without any hidden neurons. A significant improvement of the solution quality is still possible by applying NEAT for the allocation of jobs on the first stage. The results of [4] show that solving the proposed THFS as an allocation problem on the first stage can lead to solutions without any due date

Table 5. Computational efficiency of NEAT in comparison to [4]

	Runtime (s)	Hardware
ISBO	~30	CPU: 4x2.6GHz RAM: 8GB
SA	~10,800	
TS	~15,120 – 22,680	
NEAT		
Training	~300	CPU: 4x2.6GHz RAM: 8GB
Application	~0.1 – 0.2	

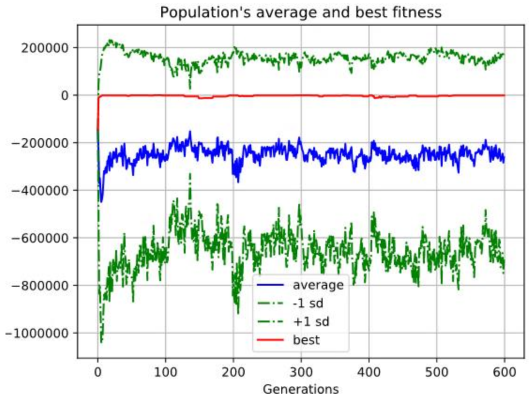

Figure 3. Development of the population's fitness in 600 generations[2]

violations and with less than half of the number of major setups.

Concerning the runtime of SA and TS, we must point out that both approaches normally converge to good solutions in a much shorter time. In a following paper, Aurich et al. were able reduce the computational time of SA to less than 45 minutes and of TS to less than 30 minutes by integrating both metaheuristics inside the simulation model [23].

6. Conclusion

In this paper, we presented a novel strategy to solve a two-stage hybrid flow shop scheduling problem with family setup times. Instead of searching after an optimal solution for any new problem instance, our approach aims to search a machine-learning model, which approximates a high-quality solution strategy for the problem in general. As a result, the runtime for determining a solution only grows linearly with the problem complexity, which leads to an outstanding computational efficiency. In terms of solution quality, the approach can compete with metaheuristics like Simulated Annealing and Tabu Search, although our best performing neural network was not able to avoid due date violations for all datasets.

In future work, we want to apply our concept to the presented THFS for solving the allocation problem on the first stage and for the combined allocation and sequencing problem without permutation constrained. Both problems provide a larger solution space, which in turn allows a broader search for higher solution quality. A successful application on the allocation problem would mean that the concept could be applied on a large number of NP-hard combinatorial optimization problems, which can be divided into a set of allocation and sequencing problems (e.g. job-shop scheduling problems, vehicle routing problems, etc.).

However, so far it is also an open question, how our approach will perform on a larger number of problem instances that show significant differences in their statistical properties. Therefore, it will be necessary to conduct further experiments that evaluates the performance and reliability of our concept.

6. References

[1] H. Emmons, and G. Vairaktarakis, 2013, Flow shop scheduling. Theoretical results, algorithms, and applications, Springer, New York.

[2] J.N.D. Gupta, 1988, "Two-stage, hybrid flowshop scheduling problem", Journal of the Operational Research Society 39 (4), pp. 359–364.

[3] R. Ruiz, and J.A. Vázquez-Rodríguez, 2010, "The hybrid flow shop scheduling problem", European Journal of Operational Research 205 (1), pp. 1–18.

[4] P. Aurich, A. Nahhas, T. Reggelin, and J. Tolujew, 2016, "Simulation-based optimization for solving a hybrid flow shop scheduling problem", Proceedings of the 2016 Winter Simulation Conference WSC'16, IEEE, Piscataway, pp. 2809–2819.

[5] C.R. Reeves, 2010, "Genetic Algorithms". In: M. Gendreau, and J.-Y. Potvin (eds.), Handbook of Metaheuristics, Springer Science + Business Media, New York, pp. 109–140.

[6] K.O. Stanley, and R. Miikkulainen, 2002, "Efficient evolution of neural network topologies", Proceedings of the 2002 Congress on Evolutionary Computation CEC'02, IEEE, Piscataway, pp. 1757–1762.

[7] K.O. Stanley, and R. Miikkulainen, 2002, "Evolving neural networks through augmenting topologies", Evolutionary computation 10 (2), pp. 99–127.

[8] A. McIntyre, M. Kallada, C.G. Miguel, and C.F. da Silva, 2018, neat-python, https://github.com/CodeReclaimers/neat-python.

[9] X. Mao, A. ter Mors, N. Roos, and C. Witteveen, 2007, Using neuro-evolution in aircraft deicing scheduling, https://dke.maastrichtuniversity.nl/nico.roos/wp-content/uploads/2016/01/MMRW07ALAMAS.pdf.

[2] Note: The value range of the ordinate axis is negative, because we formulated a negative objective function to consider the minimization of the total tardiness as maximization problem. We note that neat-python performs better, when the fitness criterion shall be maximized. The abbreviation "sd" in the legend stands for standard deviation

[10] F.J. Gomez, D. Burger, and R. Miikkulainen, 2001, “A neuro-evolution method for dynamic resource allocation on a chip multiprocessor”, Proceedings of the 2001 International Joint Conference on Neural Networks IJCNN'01, IEEE, Piscataway, pp. 2355–2360.

[11] S. Whiteson, 2005, “Improving reinforcement learning function approximators via neuroevolution”, Proceedings of the 4th international Joint Conference on Autonomous Agents and Multiagent Systems AAMAS’05, ACM, New York, pp. 1386–1387.

[12] W. Zhang, and T.G. Dietterich, 1995, “A reinforcement learning approach to job-shop scheduling”, In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Mateo, pp. 1114–1120.

[13] M.E. Aydin, and E. Öztemel, 2000, “Dynamic job-shop scheduling using reinforcement learning agents”, Robotics and Autonomous Systems 33 (2-3), pp. 169–178.

[14] Y.-C. Wang and J.M. Usher, 2005, “Application of reinforcement learning for agent-based production scheduling”, Computers & Industrial Engineering 125, pp. 73–82.

[15] Y.-R. Shiue, K.-C. Lee, and C.-T. Su, 2018, “Real-time scheduling for a smart factory using a reinforcement learning approach”, Computers & Industrial Engineering 125, pp. 604–614.

[16] C.D. Paternina-Arboleda, and T.K. Das, 2005, “A multi-agent reinforcement learning approach to obtaining dynamic control policies for stochastic lot scheduling problem”, Simulation Modelling Practice and Theory 13 (5), pp. 389–406.

[17] S. Qu, J. Wang, S. Govil, and J.O. Leckie, 2016, “Optimized adaptive scheduling of a manufacturing process system with multi-skill workforce and multiple machine types: An ontology-based, multi-agent reinforcement Learning Approach”, Procedia CIRP 57, pp. 55–60.

[18] N. Stricker, A. Kuhnle, R. Sturm, and S. Friess, 2018, “Reinforcement learning for adaptive order dispatching in the semiconductor industry”, CIRP Annals 67 (1), pp. 511–514

[19] B. Waschneck, A. Reichstaller, L. Belzner, T. Altenmüller, T. Bauernhansl, A. Knapp, and A. Kyek, 2018, “Optimization of global production scheduling with deep reinforcement learning”, Procedia CIRP 72, pp. 1264–1269.

[20] P. Csaszar, T.M. Tirpak, and P.C. Nelson, 2000, "Optimization of a high speed placement machine using tabu search algorithms", Annals of Operations Research 96, pp. 125–147.

[21] K.R. Baker, and D. Trietsch, 2009, Principles of sequencing and scheduling. Oxford: Wiley-Blackwell

[22] R. van der Ham, 2018, “salabim: Discrete event simulation and animation in Python”, Journal of Open Source Software 3 (27), pp. 767–768.

[23] P. Aurich, A. Nahhas, T. Reggelin, and M. Krist, 2017, “Simulation-based optimization of a four stage hybrid flow shop with sequence-dependent setup times and availability constraints”, Proceedings of the 16th International Conference on Modeling and Applied Simulation MAS 2017, Dime Università di Genova, Genova, pp. 144-152